chapter

15

Testing and Debugging

15.1 Introduction

In December 1999, NASA mission controllers lost contact with the Mars Polar Lander as it approached the Martian surface. The Mars Polar Lander was on a mission to study the southern polar region of the Red Planet. Contact was never reestablished, and NASA announced that the spacecraft most probably crashed onto the planet's surface during the landing process. After evaluating the situation, investigators concluded that the likely cause was faulty control software that prematurely caused the on-board engines to shut down when the probe was 40 meters above the surface rather than when the probe had actually landed. The physical complexities of sending probes into space is astounding, and the software systems that control these spacecraft are no less complex. Software is as integral to a system as any mechanical or electrical subsystem, and all the more difficult to make correct because it is "invisible." It cannot be visually observed as easily as, say, a propulsion system or landing system.

Software is everywhere today. It is in your cell phone, in your automobile—even the text of this book was processed by numerous lines of software before appearing in front of you on good old-fashioned printed pages. Because software plays a vital and critical part in our world, it is important that this software behave correctly according to specification. Designing working programs is not automatic. Programs are not correct by construction. That is, just because a program is written does not mean that it functions correctly. We must test and debug it as thoroughly as possible before we can deem it to be complete.

Programmers often spend more time debugging their programs than they spend writing them. A general observation made by experts is that an experienced programmer spends as much time debugging code as he/she does writing it. Because of this inseparable relation between writing code and testing and debugging it, we introduce you to some basic concepts in testing and debugging in this chapter.

Testing is the process of exposing bugs, and debugging is the process of fixing them. Testing a piece of code involves subjecting it to as many input conditions as possible, in order to stress the software into revealing its bugs. For example, in testing the function `ToUpper` from the previous chapter (recall that this function returns the uppercase version of an alphabetic character passed as a parameter), we might want to pass every possible ASCII value as an input parameter and observe the function's output in order to determine if the function behaves according to specification. If the function produces incorrect output for a particular input, then we've discovered a bug. It is better to find the bug while the code is still in development than to have an unsuspecting user stumble on the bug inadvertently. It would have been better for the NASA software engineers to find the bug in the Mars Polar Lander on the surface of the earth rather than encounter it 40 meters above the surface of Mars.

Using information about a program and its execution, a programmer can apply common sense to deduce where things are going awry. Debugging a program is a bit like solving a puzzle. Like a detective at a crime scene, a programmer must examine the available clues in order to track down the source of the problem. Debugging code is significantly easier if you know how to gather information about the bug—such as the value of key variables during the execution of the program—in a systematic way.

In this chapter, we describe several techniques you can use to find and fix bugs within a program. We first describe some broad categories of errors that can creep into programs. We then describe testing methods for quickly finding these errors. We finally describe some debugging techniques for isolating and repairing these errors, and we provide some defensive programming techniques to minimize the bugs in the code you write.

15.2 Types of Errors

To better understand how to find and fix errors in programs, it is useful to get a sense of the types of errors that can creep into the programs we write. There are three broad categories of errors that you are likely to encounter in your code. *Syntactic errors* are the easiest to deal with because they are caught by the compiler. The compiler notifies us of such errors when it attempts to translate the source code into machine code, often pointing out exactly in which line the error occurred. *Semantic errors*, on the other hand, are problems that can often be very difficult to repair. They occur when the program is syntactically correct but does not behave exactly as we expected. Both syntactic and semantic errors are generally typographic errors: these occur when we type something we did not mean to type. *Algorithmic errors* are errors in which our approach to solving a

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int i
6      int j;
7
8      for (i = 0; i <= 10; i++) {
9          j = i * 7;
10         printf("%d x 7 = %d\n", i, j);
11     }
12 }
```

Figure 15.1 This program contains a syntactic error

problem is wrong. They are often hard to detect and, once detected, can be very hard to fix.

15.2.1 Syntactic Errors

In C, syntactic errors (or *syntax errors* or *parse errors*) are always caught by the compiler. These occur when we ask the compiler to translate code that does not conform to the C specification. For instance, the code listed in Figure 15.1 contains a syntax error, which the compiler will flag when the code is compiled.

The declaration for the variable `i` is missing a semicolon. As a novice C programmer, missing semicolons and variable declarations will account for a good number of the syntax errors you will encounter. The good news is that these types of errors are easy to find, because the compiler detects them, and are easy to fix, because the compiler indicates where they occur. The real problems start once the syntax errors have been fixed and the harder semantic and algorithmic errors remain.

15.2.2 Semantic Errors

Semantic errors are similar to syntactic errors. They occur for the same reason: Our minds and our fingers are not completely coordinated when typing in a program. Semantic errors do not involve incorrect syntax; therefore, the program gets translated and we are able to execute it. It is not until we analyze the output that we discover that the program is not performing as expected. Figure 15.2 lists an example of the same program as Figure 15.1 with a simple semantic error (the syntax error is fixed). The program should print out a multiplication table for the number 7.

Here, a single execution of the program reveals the problem. Only one entry of the multiplication table is printed. You should be able to deduce, given your knowledge of the C programming language, why this program behaves incorrectly. Why is `11 x 7 = 70` printed out? This program demonstrates something called a *control flow* error. Here, the program's control flow, or the order in which statements are executed, is different than we intended.

The code listed in Figure 15.3 contains a common, but tricky semantic error involving local variables. This example is similar to the factorial program we discussed in Section 14.2.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int i;
6    int j;
7
8    for (i = 0; i <= 10; i++)
9      j = i * 7;
10     printf("%d x 7 = %d\n", i, j);
11 }
```

Figure 15.2 A program with a semantic error

This program calculates the sum of all integers less than or equal to the number input from the keyboard (i.e., it calculates $1 + 2 + 3 + \ldots + n$). Try executing this program and you will notice that the output is not what you would expect. Why doesn't it work properly? Hint: Draw out the run-time stack for an execution of this program.

Semantic errors are particularly troubling because they often go undetected by both the compiler *and* the programmer until a particular set of inputs triggers

```
1  #include <stdio.h>
2
3  int AllSum(int n);
4
5  int main()
6  {
7    int in;                          /* Input value                */
8    int sum;                         /* Value of 1+2+3+..+n        */
9
10   printf("Input a number: ");
11   scanf("%d", &in);
12
13   sum = AllSum(in);
14   printf("The AllSum of %d is %d\n", in, sum);
15 }
16
17
18 int AllSum(int n)
19 {
20   int result;                      /* Result to be returned      */
21   int i;                           /* Iteration count            */
22
23   for (i = 1; i <= n; i++)         /* This calculates sum        */
24     result = result + i;
25
26   return result;                   /* Return to caller           */
27 }
```

Figure 15.3 A program with a bug involving local variables

the error. Refer to the `AllSum` program in Figure 15.3, but repair the previous semantic error and notice that if the value passed to `AllSum` is less than or equal to 0 or too large, then `AllSum` may return an erroneous result because it has exceeded the range of the integer variable `result`. Fix the previous bug, compile the program, and input a number smaller than 1 and you will notice another bug.

Some errors are caught during execution because an illegal action is performed by the program. Almost all computer systems have safeguards that prevent a program from performing actions that might affect other unrelated programs. For instance, it is undesirable for a user's program to modify the memory that stores the operating system or to write a control register that might affect other programs, such as a control register that causes the computer to shut down. When such an illegal action is performed by a program, the operating system terminates its execution and prints out a *run-time error* message. Modify the `scanf` statement from the `AllSum` example to the following:

```
scanf("%d", in);
```

In this case, the ampersand character, `&`, as we shall see in Chapter 16, is a special operator in the C language. Omitting it here causes a run-time error because the program has attempted to modify a memory location to which it does not have access. We will look at this example and the reasons for the error in more detail in later chapters.

15.2.3 Algorithmic Errors

Algorithmic errors are the result of an incorrect program design. That is, the program itself behaves exactly as we designed, but the design itself was flawed. These types of errors can be hidden; they may not appear until many trials of the program have been run. Even when they are detected and isolated, they can be very hard to repair. The good news is that these types of errors can often be reduced and even eliminated by proper planning during the design phase, before any code is written.

An example of a program with a simple algorithmic flaw is provided in Figure 15.4. This code takes as input the number of a calendar year and determines if that year is a leap year or not.

At first glance, this code appears to be correct. Leap years do occur every four years. However they are skipped at the turn of every century, **except** every fourth century (i.e., the year 2000 was a leap year, but 2100, 2200, and 2300 will not be). The code works for almost all years, except those falling into these exceptional cases. We categorize this as an algorithmic error, or design flaw.

Another example of an algorithmic error also involving dates is the infamous Year 2000 computer bug, or Y2K bug. Many computer programs minimize the amount of memory required to store dates. They use enough bits to store only the last two digits of the year, and no more. Thus, the year 2000 is indistinguishable from the year 1900 (or 1800 or 2100 for that matter). This presented a problem during the recent century crossover on December 31, 1999. Say, for example, you had checked out a book from the university library in late 1999 and it was due back sometime in early 2000. If the library's computer system suffered from the

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int year;
6
7    printf("Input a year (i.e., 1996): ");
8    scanf("%d", &year);
9
10   if (year % 4 == 0)
11      printf("This year is a leap year"\n);
12   else
13      printf("This year is not a leap year"\n);
14 }
```

Figure 15.4 This program to determine leap years has an algorithmic bug

Y2K bug, you would have gotten an overdue notice in the mail with some hefty fines listed on it. As a consequence, a lot of money and effort were devoted to tracking down Y2K-related bugs before January 1, 2000 rolled around.

15.3 Testing

There is an adage among seasoned programmers that any line of code that is untested is probably buggy. Good testing techniques are crucial to writing good software. What is testing? With testing, we basically put the software through trials where input patterns are applied (in order to mimic what the software might see during real operation) and the output of the program is checked for correctness. Real-world software might undergo millions of trials before it is released.

In an ideal world, we could test a program by examining its operation under every possible input condition. But for a program that is anything more than trivial, testing for every input combination is impossible. For example, if we wanted to test a program that finds prime numbers between integers A and B, where A and B are 32-bit input values, there are $(2^{32})^2$ possible input combinations. Even if we could run 1 million trials in 1 second, it would still take half a million years to completely test the program. Clearly, testing each input combination is not an option. So which input combinations do we test with? We could randomly pick inputs in hopes that some of those random patterns will expose the program's bugs. Software engineers typically rely on more systematic ways of testing their code. In particular, black-box testing is used to check if a program meets its specifications, and white-box testing targets various facets of the program's implementation in order to provide some assurance that every line of code is tested.

15.3.1 Black-Box Testing

With black-box testing, we examine if the program meets its input and output specifications, disregarding the internals of the program. That is, with black-box testing, we are concerned with what the program does and not how it does it. For

example, a black-box test of the program `AllSum` in Figure 15.3 might involve running the program, typing an input number, and comparing the resulting output to what you calculated by hand. If the two do not match, then either the program contains a bug or your arithmetic skills are shoddy. We might continue attempting trials until we are reasonably confident that the program is functional.

For testing larger programs, the testing process is *automated* in order to run more tests per unit time. That is, we construct another program to automatically run the original program, provide some random inputs, check that the output meets specifications, and repeat. With such a process, we can clearly run many more trials than we could if a person performed each trial.

In order to automate the black-box process, however, we need a way to automatically test whether the program's output was correct or incorrect. Here, we might need to construct a checker program that is different than the original program but performs a similar computation. If the original and checker programs had the same bug, it would go undetected by the black-box testing process. For this reason, black-box testers who write checker programs are often not permitted to see the code within the black box they are testing so that we get a truly independent version of the checker.

15.3.2 White-Box Testing

For larger software systems, black-box testing is not enough. With black-box testing, it is not possible to know which lines of code have been tested and which have not, and therefore, according to the adage stated previously, all are presumed to be buggy. Black-box testing is sometimes difficult when the input or output specification of a program is not concrete. For example, black-box testing of an audio player (such as an MP3 player) might be difficult because of the inexact nature of the output. Also, black-box testing can only start once the software is complete—the software must compile and must meet some part of the specification in order to be tested.

Software engineers supplement black-box testing with white-box tests. White-box tests isolate various internal components of the software, and test whether the components conform to their intended design. For example, testing to see that each function performs correctly according to the design is a white-box test. How we divide a program into functions is part of its implementation and not its specification. We can apply the same type of testing to loops and other constructs within a function.

How might a white-box test be constructed? For many tests, we might need to modify the code itself. For example, in order to see whether a function is working correctly, we might add extra code to call the function a few extra times with different inputs and check the outputs. We might add extra `printf` statements to the code with which we can observe values of internal variables to see if things are working as expected. Once the code is complete and ready for release, these `printf` statements can be removed.

A common white-box testing technique is the use of error-detecting code strategically placed within a program. This code might check for conditions that indicate that the program is not working correctly. When an incorrect situation is

detected, the code prints out a warning message, displays some relevant information about the situation, or causes the program to prematurely terminate. Since this error-detecting code *asserts* that certain conditions hold during program execution, we generally call these checks *assertions*.

For example, assertions can be used to check whether a function returns a value within an expected range. If the return value is out of this range, an error message is displayed. In the following example, we are checking whether the calculation performed by the function `IncomeTax` is within reasonable bounds. As you can deduce from this code fragment, this function calculates the income tax based on a particular income provided as a parameter to it. We do not pay more tax than we collect in income (fortunately!), and we never pay a negative tax. Here if the calculation within `IncomeTax` is incorrect, a warning message will be displayed by the assertion code.

```
tax = IncomeTax(income);

if (tax < 0 || tax > income)
   printf("Error in function IncomeTax!\n");
```

A thorough testing methodology requires the use of both black-box and white-box tests. It is important to realize that white-box tests alone do not cover the complete functionality of the software—even if all white-box tests pass, there might be a portion of the specification that is missing. Similarly, black-box tests alone do not guarantee that every line of code is tested.

15.4 Debugging

Once a bug is found, we start the process of repairing it, which can often be more tricky than finding it. Debugging an error requires full use of our reasoning skills: We observe a symptom of the error, such as bad output, and we might even have some other information, such as the place in the code where the error occurred, and from this limited information, we will need to use deduction to isolate the source of the error. The key to effective debugging is being able to quickly gather relevant information that will lead to identifying the bug, similar to the way a detective might gather evidence at a crime scene or the way a physician might perform a series of tests in order to diagnose a sick patient's illness.

There are a number of ways you can gather more information in order to diagnose a bug, ranging from ad hoc techniques that are quick and dirty to more systematic techniques that involve the use of software debugging tools.

15.4.1 Ad Hoc Techniques

The simplest thing to do once you realize that there is a problem with your program is to visually inspect the source code. Sometimes the nature of the failure tips you off to the region of the code where the bug is likely to exist. This technique is fine if the region of source code is small and you are very familiar with the code.

Another simple technique is to insert statements within the code to print out information during execution. You might print out, using `printf` statements, the values of important variables that you think will be useful in finding the bug. You can also add `printf` statements at various points within your code to see if the control flow of the program is working correctly. For example, if you wanted to quickly determine if a counter-controlled loop is iterating for the correct number of iterations, you could place a `printf` statement within the loop body. For simple programs, such ad hoc techniques are easy and reasonable to use. Large programs with intricate bugs require the use of more heavy-duty techniques.

15.4.2 Source-Level Debuggers

Often ad hoc techniques cannot provide enough information to uncover the source of a bug. In these cases, programmers often turn to a *source-level debugger* to isolate a bug. A source-level debugger is a tool that allows a program to be executed in a controlled environment, where all aspects of the execution of the program can be controlled and examined by the programmer. For example, a debugger can allow us to execute the program one statement at a time and examine the values of variables (and memory locations and registers, if we so choose) along the way. Source-level debuggers are similar to the LC-3 debugger that we described in Chapter 6, except that a source-level debugger operates in relation to high-level source code rather than LC-3 machine instructions.

For a source-level debugger to be used on a program, the program must be compiled such that the compiler augments the executable image with enough additional information for the debugger to function properly. Among other things, the debugger will need information from the compilation process in order to map every machine language instruction to its corresponding statement in the high-level source program. The debugger also needs information about variable names and their locations in memory (i.e., the symbol table). This is required so that a programmer can examine the value of any variable within the program using its name in the source code.

There are many source-level debuggers available, each of which has its own user interface. Different debuggers are available for UNIX and Windows, each with its own flavor of operation. For example, `gdb` is a free source-level debugger available on most UNIX-based platforms. All debuggers support a core set of necessary operations required to probe a program's execution, many of which are similar to the debugging features of the LC-3 debugger. So rather than describe the user interface for any one particular debugger, in this section we will describe the core set of operations that are universal to any debugger.

The core debugger commands fall into two categories: those that let you control the execution of the program and those that let you examine the value of variables and memory, etc. during the execution.

Breakpoints

Breakpoints allow us to specify points during the execution of a program when the program should be temporarily stopped so that we can examine or modify the

state of the program. This is useful because it helps us examine the program's execution in the region of the code where the bug occurs.

For example, we can add a breakpoint at a particular line in the source code or at a particular function. When execution reaches that line, program execution is frozen in time, and we can examine everything about that program at that particular instance. How a breakpoint is added is specific to the user interface of the debugger. Some allow breakpoints to be added by clicking on a line of code. Others require that the breakpoint be added by specifying the line number through a command prompt.

Sometimes it is useful to stop at a line only if a certain condition is true. Such conditional breakpoints are useful for isolating specific situations in which we suspect buggy behavior. For example, if we suspect that the function `PerformCalculation` works incorrectly when its input parameter is 16, then we might want to add a breakpoint that stops execution only when `x` is equal to 16 in the following code:

```
for (x = 0; x < 100; x++)
   PerformCalculation(x);
```

Alternatively, we can set a *watchpoint* to stop the program at any point where a particular condition is true. For example, we can use a watchpoint to stop execution whenever the variable `LastItem` is equal to 4. This will cause the debugger to stop execution at any statement that causes `LastItem` to equal 4. Unlike breakpoints, watchpoints are not associated with any single line of the code but apply to every line.

Single-Stepping

Once the debugger reaches a breakpoint (or watchpoint), it temporarily suspends program execution and awaits our next command. At this point we can examine program state, such as values of variables, or we can continue with execution.

It is often useful to proceed from a breakpoint one statement at time—a process referred to as *single-stepping*. The LC-3 debugger has a command that executes a single LC-3 instruction and similarly a source-level debugger that allows execution to proceed one statement at a time. The single-step command executes the current source line and then suspends the program again. Most debuggers will also display the source code in a separate window so we can monitor where the program has currently been suspended. Single-stepping through a program is very useful, particularly when executing the region of a program where the bug is suspected to exist. We can set a breakpoint near the suspected region and then check the values of variables as we single-step through the code.

A common use of single-stepping is to verify that the control flow of the program does what we expect. We can single-step through a loop to verify that it performs the correct number of iterations or we can single-step through an `if-else` to verify that we have programmed the condition correctly.

Variations of single-stepping exist that allow us to skip over functions, or to skip to the last iteration of a loop. These variations are useful for skipping over

code that we do not suspect to contain errors but are in the execution path between a breakpoint and the error itself.

Displaying Values

The art of debugging is about gathering the information required to logically deduce the source of the error. The debugger is the tool of choice for gathering information when debugging large programs. While execution is suspended at a breakpoint, we can gather information about the bug by examining the values of variables related to the suspected bug. Generally speaking, we can examine all execution states of the program at the breakpoint. We can examine the values of variables, memory, the stack, and even the registers. How this is done is debugger specific. Some debuggers allow you to use the mouse to point to a variable in the source code window, causing a pop-up window to display the variable's current value. Some debuggers require you to type in a command indicating the name of the variable you want to examine.

We encourage you to familiarize yourself with a source-level debugger. At the end of this chapter, we provide several problems that you can use to gain some experience with this useful debugging tool.

15.5 Programming for Correctness

Knowing how to test and debug your code is a prerequisite for being a good programmer. Great programmers know how to avoid many error-causing situations in the first place. Poor programming practices cause bugs. Being aware of some defensive programming techniques can help reduce the amount of time required to get a piece of code up and running. The battle against bugs starts before any line of code is written. Here, we provide three general methods for catching errors even before they become errors.

15.5.1 Nailing Down the Specification

Many bugs arise from poor or incomplete program specifications. Specifications sometimes do not cover all possible operating scenarios, and thus they leave some conditions open for interpretation by the programmer. For example, recall the factorial example from Chapter 14: Figure 14.2 is a program that calculates the factorial of a number typed at the keyboard. You can imagine that the specification for the program might have been "Write a program to take an integer value from the keyboard and calculate its factorial." As such, the specification is incomplete. What if the user enters a negative number? Or zero? What if the user enters a number that is too large and results in an overflow? In these cases, the code as written will not perform correctly, and it is therefore buggy. To fix this, we need to modify the specification of the program to allow the program to indicate an error if the input is less than or equal to zero, or if the input is such that $n! > 2^{31}$, implying n must be less than or equal to 31. In the code that follows we have added an input range check to the `Factorial` function from Chapter 14. Now

the function prints a warning message and returns a −1 if its input parameter is out of the correct operating range.

```
1   int Factorial(int n)
2   {
3     int i;                    /* Iteration count          */
4     int result = 1;           /* Initialized result       */
5
6     /* Check for legal parameter values */
7     if (n < 1 || n > 31) {
8       printf("Bad input. Input must be >= 1 and <= 31.\n");
9       return -1;
10    }
11
12    for (i = 1; i <= n; i++)  /* Calculates factorial     */
13      result = result * i;
14
15    return result;            /* Return to caller         */
16  }
```

15.5.2 Modular Design

Functions are useful for extending the functionality of the programming language. With functions we can add new operations and constructs that are helpful for a particular programming task. In this manner, functions enable us to write programs in a modular fashion.

Once a function is complete, we can test it independently in isolation (i.e., as a white-box test) and determine that it is working as we expect. Since a typical function performs a smaller task than the complete program, it is easier to test than the entire program. Once we have tested and debugged each function in isolation, we will have an easier chance getting the program to work when everything is integrated.

This modular design concept of building a program out of simple, pretested, working components is a fundamental concept in systems design. In subsequent chapters we will introduce the concept of a *library*. A library is a collection of pretested components that all programmers can use in writing their code. Modern programming practices are heavily oriented around the use of libraries because of the benefits inherent to modular design. We design not only software, but circuits, hardware, and various other layers of the computing system using a similar modular design philosophy.

15.5.3 Defensive Programming

All seasoned programmers have techniques to prevent bugs from creeping into their code. They construct their code in a such a way that those errors that they

suspect might affect the program are eliminated by design. That is, they program *defensively*. We provide a short list of general defensive programming techniques that you should adopt to avoid problems with the programs you write.

- Comment your code. Writing comments makes you think about the code you've written. Code documentation is not only a way to inform others about how your code works, but also is a process that makes you reflect on and reconsider your code. During this process you might discover that you forgot a special case or operating condition that will ultimately break your code.
- Adopt a consistent coding style. For instance, aligning opening and closing braces will let you identify simple semantic errors associated with missing braces. Along these lines, also be consistent in variable naming. The name of a variable should convey some meaningful information about the value the variable contains.
- Avoid assumptions. It is tempting to make simple, innocent assumptions when writing code, but these can ultimately lead to broken code. For example, in writing a function, we might assume that the input parameter will always be within a certain range. If this assumption is not grounded in the program's specification, then the possibility for an error has been introduced. Write code that is free of such assumptions—or at least use assertions and spot checks to indicate when the assumptions do not hold.

- Avoid global variables. While some experienced programmers rely heavily on global variables, many software engineers advocate avoiding them whenever possible. Global variables can make some programming tasks easier. However, they often make code more difficult to understand, and extend, and when a bug is detected, harder to analyze.
- Rely on the compiler. Most good compilers have an option to carefully check your program for suspicious code (for example, an uninitialized variable) or commonly misapplied code constructs (for example, using the assignment operator `=` instead of the equality operator `==`). While these checks are not thorough, they do help identify some commonly made programming mistakes. If you are use the `gcc` compiler, use `gcc -Wall` to enable all warning messages from the compiler.

The defensive techniques mentioned here are particular to the programming concepts we've already discussed. In subsequent chapters, after we introduce new programming concepts, we also discuss how to use defensive techniques when writing programs that use them.

15.6 Summary

In this chapter, we presented methodologies for finding and fixing bugs within your code. Modern systems are increasingly reliant on software, and modern software is often very complex. In order to prevent software bugs from often rendering our cell phones unusable or from occasionally causing airplanes to

crash, it is important that software tightly conform to its specifications. The key concepts that we covered in this chapter are:

- **Testing.** Finding bugs in code is not easy, particularly when the program is large. Software engineers use systematic testing to find errors in software. Black-box testing is done to validate that the behavior of a program conforms to specification. White-box testing targets the structure of a program and provides some assurance that every line of code has undergone some level of testing.

- **Debugging.** Debugging an error requires the ability to take the available information and deduce the source of the error. While ad hoc techniques can provide us with a little additional information about the bug, the source-level debugger is the software engineering tool of choice for most debugging tasks. Source-level debuggers allow a programmer to execute a program in a controlled environment and examine various values and states within the program during execution.

- **Programming for correctness.** Experienced programmers try to avoid bugs even before the first line of code is written. Often, the specification of the program is the source of bugs, and nailing down loose ends will help eliminate bugs after the code has been written. Modular design involves writing a larger program out of simple pretested functions and helps reduce the difficulty in testing a large program. Following a defensive programming style helps reduce situations that lead to buggy code.

Exercises

15.1 The following programs each have a single error that prevents them from operating as specified. With as few changes as possible, correct the programs. They all should output the sum of the integers from 1 to 10, inclusive.

a.
```
#include <stdio.h>
int main()
{
    int i = 1;
    int sum = 0;

    while (i < 11) {
        sum = sum + i;
        ++i;
        printf("%d\n", sum);
    }
}
```

b.
```
#include <stdio.h>
int main()
{
    int i;
    int sum = 0;

    for (i = 0; i >= 10; ++i)
        sum = sum + i;
    printf("%d\n", sum);
}
```

c.
```
#include <stdio.h>
int main()
{
    int i = 0;
    int sum = 0;

    while (i <= 11)
        sum = sum + i++;
    printf("%d\n", sum);
}
```

d.
```
#include <stdio.h>
int main()
{
    int i = 0;
    int sum = 0;

    for (i = 0; i <= 10;)
        sum = sum + ++i;
    printf("%d\n", sum);
}
```

15.2 The following program fragments have syntax errors and therefore will not compile. Assume that all variables have been properly declared. Fix the errors so that the fragments will not cause compiler errors.

a.
```
i = 0;
j = 0;
while (i < 5)
{
   j = j + 1;
   i = j >> 1
}
```

b.
```
if (cont == 0)
   a = 2;
   b = 3;
else
   a = -2;
   b = -3;
```

c.
```
#define LIMIT 5;

if (LIMIT)
   printf("True");
else
   printf("False");
```

15.3 The following C code was written to find the minimum of a set of positive integers that a user enters from the keyboard. The user signifies the end of the set by entering the value −1. Once all the numbers have been entered and processed, the program outputs the minimum. However, the code contains an error. Identify and suggest ways to fix the error. Use a source-level debugger, if needed, to find it.

```
#include <stdio.h>
int main()
{
   int smallestNumber = 0;
   int nextInput;

   /* Get the first input number */
   scanf("%d", &nextInput);

   /* Keep reading inputs until user enters -1 */
   while (nextInput != -1) {
      if (nextInput < smallestNumber)
         smallestNumber = nextInput;
      scanf("%d", &nextInput);
   }
   printf("The smallest number is %d\n", smallestNumber);
}
```

15.4 The following program reads in a line of characters from the keyboard and echoes only the alphabetic, numeric, and space characters. For example, if the input were "`Let's meet at 6:00pm.`", the output should be: "`Lets meet at 600pm`". The program does not work as specified. Fix it.

```
#include <stdio.h>
int main()
{
  char echo = '0';

  while (echo != '\n') {
    scanf("%c", &echo);
    if ((echo > 'a' || echo < 'z') &&
        (echo > 'A' || echo < 'Z'))
      printf("%c", echo);
  }
}
```

15.5 Use a source-level debugger to monitor the execution of the following code:

```
#include <stdio.h>

int IsDivisibleBy(int dividend, int divisor);

int main()
{
   int i;  /* Iteration variable                    */
   int j;  /* Iteration variable                    */
   int f;  /* The number of factors of a number */


   for (i = 2; i < 1000; i++) {

      f = 0;
      for (j = 2; j < i; j++) {
         if (IsDivisibleBy(i, j))
            f++;
      }
      printf("The number %d has %d factors\n", i, f);
   }
}

int IsDivisibleBy(int dividend, int divisor)

{
   if (dividend % divisor == 0)
      return 1;
   else
      return 0;
}
```

a. Set a breakpoint at the beginning of function `IsDivisibleBy` and examine the parameter values for the first 10 calls. What are they?

b. What is the value of `f` after the inner `for` loop ends and the value of `i` equals 660?

c. Can this program be written more efficiently? Hint: Monitor the value of the arguments when the return value of `IsDivisibleBy` is 1.

15.6 Using a source-level debugger, determine for what values of parameters the function `Mystery` returns a zero.

```
#include <stdio.h>

int Mystery(int a, int b, int c);

int main()
{
   int i;          /* Iteration variable          */
   int j;          /* Iteration variable          */
   int k;          /* Iteration variable          */
   int sum = 0;    /* running sum of Mystery      */

   for (i = 100; i > 0; i--) {
     for (j = 1; j < i; j++) {
       for (k = j; k < 100; k++)
         sum = sum + Mystery(i, j, k);
     }
   }
}

int Mystery(int a, int b, int c)
{
  int out;

  out = 3*a*a + 7*a - 5*b*b + 4*b + 5*c ;

  return out;
}
```

15.7 The following program manages flight reservations for a small airline that has only one plane that has SEATS number of seats for passengers. This program processes ticket requests from the airline's website. The command *R* requests a reservation. If there is a seat available, the reservation is approved. If there are no seats, the reservation is denied. Subsequently, a passenger with a reservation can purchase a ticket using the *P* command. This means that for every *P* command, there must be a preceding *R* command; however, not every *R* will materialize into a purchased ticket. The program ends when the *X* command is entered. Following is the program, but it contains serious design errors. Identify the errors. Propose and implement a correct solution.

```
#include <stdio.h>

#define SEATS 10

int main()
{
  int seatsAvailable = SEATS;
  char request = '0';

  while (request != 'X') {
    scanf("%c", &request);

    if (request == 'R') {
      if (seatsAvailable)
        printf("Reservation Approved!\n");
      else
        printf("Sorry, flight fully booked.\n");
    }

    if (request == 'P') {
      seatsAvailable--;
      printf("Ticket purchased!\n");
    }
  }

  printf("Done! %d seats not sold\n", seatsAvailable);
}
```